LLNL-CONF-609796

Systematic Reduction of Data Movement in Algebraic Multigrid Solvers

H. Gahvari, W. Gropp, K. E. Jordan, M. Schulz, U. M. Yang

January 8, 2013

Workshop on Large-Scale Parallel Processing to be held at IPDPS 2013
Boston, MA, United States
May 20, 2013 through May 24, 2013

Systematic Reduction of Data Movement in Algebraic Multigrid Solvers

Hormozd Gahvari[1], William Gropp[1], Kirk E. Jordan[2], Martin Schulz[3] and Ulrike Meier Yang[3]

[1]Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801
[2]Computational Science Center, IBM TJ Watson Research Center, Cambridge, MA 02142
[3]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551

Abstract

Algebraic multigrid (AMG) solvers find wide use in scientific simulation codes. Their ideal computational complexity makes them especially attractive for solving large problems on parallel machines. However, they also involve a substantial amount of data movement, posing challenges to performance and scalability. In this paper, we present an algorithm that provides a systematic means of reducing data movement in AMG. The algorithm operates by gathering and redistributing the problem data to reduce the need to move it on the communication-intensive coarse grid portion of AMG. The data is gathered in a way that ensures data locality by keeping data movement confined to specific regions of the machine. Any decision to gather data is made systematically through the means of a performance model. This approach results in substantial speedups on a multicore cluster when using AMG to solve a variety of test problems.

I. Introduction

The rising scale of HPC systems not only requires applications to match the increased level of concurrency available in the system, but also imposes new constraints and limitations, in particular in terms of resilience and power consumption. The latter is directly tied to data movements, which are responsible for a majority of the power consumed in a system. To address these challenges and to successfully exploit future architectures, we therefore need new algorithmic approaches that specifically target the reduction of data movements and at the same time offer new avenues for resiliency.

In our work we approach this topic systematically from the algorithmic side focusing on Algebraic Multigrid (AMG) methods, a class of solvers for large, sparse linear systems of equations that finds use in a wide range of scientific applications. AMG has the ideal property of having a computational complexity that is linear in the number of unknowns, when it works well, and has shown excellent weak scaling to the size of current high-end systems, such as IBM Blue Gene/L [1] and Blue Gene/P [2]. However, for architectures with wide multicore nodes, AMG is starting to run into scaling bottlenecks, which are directly connected to its algorithmic approach. AMG obtains its optimal computation complexity by using smaller "coarse grid" problems to accelerate the solution of the original "fine grid" problem. Since the number of nonzeros per row for the matrices on the coarser levels grows, communication complexity also increases significantly, leading to a large number of messages. Making things worse, the number of nodes involved in the communication only decreases slowly, in particular on systems with wide multicore nodes, since often at least one core per node still participates at coarser grid levels. However, the large number of processes at coarser grid levels can be aggregated and the resulting set of tasks can be either executed on a subset of nodes (agglomeration approach) or several copies of this set redundantly across various subsets of nodes (redundant approach). If the strategy is applied at the correct level for the redundant approach and in general for the agglomeration approach, the communication requirements at coarser levels are significantly reduced, and in the extreme case even eliminated by aggregating the computation onto a single process.

In this paper we focus on the latter approach, the use of redundancy, since it not only reduces communication requirements, but also offers potentials for implicit resiliency due to the redundant nature of the computation. Our algorithm partitions the problem domain into chunks and distributes these chunks to subsets of the involved processes. By carefully selecting which processes get which chunks, communication can be made to occur only in localized fashion.

One of the critical aspects of this approach is the decision about the right level to switch to redundant cycling. In our approach we guide this decision at runtime based on a performance model of the AMG cycle, which we extended to include redundant cycling. This enables us to automatically tune our approach to the input set without loss of generality.

In particular, this paper makes the following contributions:

- We explore a new algorithmic approach for AMG that reduces overall communication using redundant data distributions,
- We extend an existing performance model to cover redundant data distribution schemes, and
- We use this model to dynamically determine the appropriate level for switching to the redundant scheme.

Overall, our algorithm combined with our novel model guided dynamic level adaptation is up to 3x faster when using AMG to solve a variety of realistic problem cases.

II. Algebraic Multigrid

Multigrid methods are widely used when solving large equation systems on parallel machines, since they are optimal, i.e. they can solve a linear system with N unknowns with $O(N)$ comutational work. They achieve this by performing part of the work that other solvers would perform on the original "fine grid" problem on smaller "coarse grid" problems instead. This process is known as coarse grid correction. After the application of an inexpensive smoother, such as a Jacobi or Gauss-Seidel iteration on the fine grid, which results in an approximate solution, that approximation is corrected on the next coarsest grid. The correction can be obtained either through direct solution or another round of smoothing followed by a further recursive coarse grid correction. The use of multiple grids gives the process the term multigrid; we consider here its simplest form, the V-cycle, in which the work proceeds from the finest grid to the coarsest grid and then back to the finest grid. We number the grids from finest to coarsest; if there are L grids, the finest grid is level 0, and the coarsest grid is level $L - 1$.

The first multigrid methods were geometric in nature, and were used to solve problems on structured grids. AMG extends multigrid to cover problems on unstructured grids such as finite element meshes. All that is required is a linear system $Ax = b$. The grid information is inferred from the graph of A. This requires AMG to operate in two phases. The first is a setup phase, in which the hierarchy of grids including various operators is determined. At each level, the grid points that will remain on the next coarsest grid are selected, followed by the formation of an operator which restricts the residual on that level to the next coarsest grid and a prolongation operator, which interpolates the correction back up from that grid. Here, as is often the case in practice, the restriction interpolation is chosen to be the transpose of the interpolation operator.

The solve operator for the next grid is then formed by multiplying the restriction, the solve and the interpolation operator of the current level. Once the hierarchy of grids has been set up, AMG switches to the solve phase. In the solve phase, a multigrid cycle, such as a V-cycle, is applied to the hierarchy of grids generated in the setup phase.

There are a number of implementations of AMG, which offer access to a wide variety of coarsening and interpolation schemes for generating the hierarchy of grids and a number of smoothers for use with the solve phase. In our experiments, we use the BoomerAMG solver [3] in the hypre software library [4]. We use HMIS coarsening [5] with extended+i interpolation [6] truncated to at most 4 elements per row. For 3D problems, we also use aggressive coarsening with multipass interpolation [7] on the first level. For the smoother, we use hybrid Gauss-Seidel, which is comprised of Gauss-Seidel iteration between process boundaries and Jacobi iteration across process boundaries.

The efficient implementation of multigrid methods on parallel machines has been an area of research for quite some time, leading to the development of several variants with reduced data movement. Much of this work is centered around the basic idea of accumulating and redistibuting the problem data onto a subset of the processes at a particular level of the multigrid cycle and performing the rest of the cycle on those processes only. One option to implement such a scheme is to redundantly distribute the data onto those processors. Then each one has the same data, meaning that they no longer have to communicate with each other during the redundantly treated levels. Gropp [8] found this approach to be beneficial for geometric multigrid in some cases, and significant speedups could be achieved for AMG [9], but in both cases the benefits are diminished at scale. Womble and Young [10] used a more gradual, bottom-up approach to redundancy in geometric multigrid, but did not consider data locality.

Another variant of the data gathering approach uses plain agglomeration, i.e., it involves simply concentrating the data replicated by the redundant approach onto a subset of the involved processes without replication, performing the rest of the cycle there, and then redistributing the result to the originally involved processes. Such an approach has been used by Nakajima [11] for the coarse grid solve, in Sandia's ML smoothed aggregation AMG solver [12] and by Sampath and Biros [13] for an octree-based geometric multigrid solver to deal with convergence degradation and/or load imbalance.

Our focus here is on the overall reduction of data movement. While future plans include the investigation of both the agglomeration and redundant approach, we focus first on the redundant variant, due to its potential for data recovery and improved fault tolerance in AMG. In the next section, we present our algorithm for reducing data movement in AMG, which relies on a performance model to decide the amount and level of data gathering to be performed. This model can provide the necessary information to find the best tradeoff between reduced data movement and excess computation. Additionally, our algorithm is designed to gather data within specific portions of the machine to improve data locality, further reducing the amount of data movement.

III. Algorithm

Our algorithm takes the basic redundant approach of [8] and [9], but significantly enhances scalability by distributing smaller portions of the problem data to each process. Since, as a consequence, this will not be the entire problem data, the redundant phase still requires communication, but there will be far fewer messages to send. To further reduce communication, the algorithm also allows for the data to be gathered in specific regions of the machine to improve data locality.

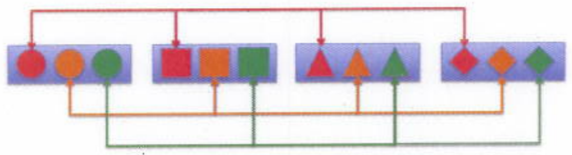

Fig. 1. Illustration of chunk data distribution with 4 chunks (blue blocks) and 12 processes (red, orange, and green shapes). Each chunk is owned by 3 processes depicted as having the same shape. The communication pattern can be regularized as shown by the arrows, with processes in a color group communicating only with each other.

We call our algorithm the chunks algorithm because the data is gathered into a set of chunks that only have to communicate with other chunks, but do not communicate within a chunk as illustrated in Figure 1 for 4 chunks with 3 processes each. For the agglomeration approach, each chunk would only consist of one process. As one of the key contributions, we tie our algorithm to a performance model that allows the algorithm to dynamically decide when to switch to redundancy to best improve performance. The model can also be applied to the agglomeration approach.

A. Algorithm Details

In our implementation, we keep coarse grid points on the same process as their corresponding fine grid points until the level, on which the chunks algorithm is applied. The chunks algorithm is then implemented as follows: During the AMG setup phase, we form two sets of MPI communicators: collective communicators, which gather data to form chunks, and point-to-point communicators, to handle communication between chunks. We create one collective communicator for each chunk, and one point-to-point communicator for each cluster, which we define to be a group of processes in different chunks that will need to exchange data during the solve phase. After the communicators are formed, all processes within a chunk perform `MPI_Allgatherv` operations to acquire the matrix data for their chunk. Then, new parallel matrices are formed over processes in each cluster. Once this is complete, each matrix is treated as the finest level operator for a new coarse solver object, and the parallel AMG setup routine is called for each of the new matrices. If there is just one chunk, then it is the basic redundant case. In this case, the collective communicator is `MPI_COMM_WORLD` (or a smaller communicator consisting of just the active processes if some have dropped out), and there is no point-to-point communication.

In the solve phase, we perform additional `MPI_Allgatherv` operations in the cycle during the switchover to the redundant phase to split and reorganize the right hand side and the solution vector into chunks. The solve cycle is then called on the coarse solver object and the reorganized right hand side and solution vector. Once that portion of the cycle is complete, the appropriate pieces of the result are copied from the chunks solution vector to the non-chunks solution vector. This step requires no communication, as the processes in each chunk will already have all data needed stored locally.

To maximize the benefits of the chunks algorithm, we must place data in a way that keeps communication within well-defined units of the machine, as this preserves locality. Such considerations are going to be even more important on future machines, with data movement becoming more and more expensive relative to computation in terms of both performance and power. How to best do this depends on the mapping of processes to nodes in the machine. In Figure 2, we illustrate mapping strategies for 12 processes and 4 chunks.

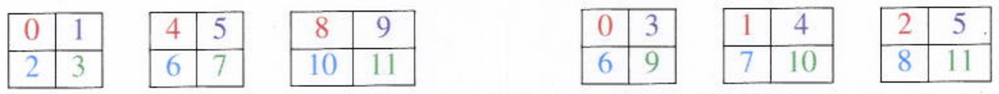

Fig. 2. Assignment of processes to collective and point-to-point communicators for chunks algorithm for the case of 12 processes and 4 chunks using a block mapping (left) or a cyclic mapping (right). Processes with the same color are in the same collective communicator (i.e. own the same chunk of data); processes in the same box are in the same point-to-point communicator.

B. Data Gathering Automation with the Performance Model

We use a performance model of the AMG solve cycle to automate the decision to switch to the chunks algorithm. Our model is based on a simple latency-bandwidth model for communication and then adds a communication distance term and penalties to take into account multicore issues and limited bandwidth. It is able to capture the performance of AMG on a number of different machines and under MPI-only and hybrid MPI/OpenMP programming models while making use of machine parameters that can be determined from simple measurements and/or the topology of the interconnect; the details of how are in our past work on this subject [14]–[16].

When deciding whether or not to switch to chunks at a given level l, we use the model to predict T_{noswitch}, the time spent at this level if we did not switch, and T_{switch}, the time spent if we switch. If the latter is less, then we switch. There is no benefit from switching at a lower level because switching right away would capture the benefit from the later switch plus the benefit from switching at the current level.

To determine T_{noswitch}, we use the model to predict the time of five matrix-vector multiplications using the existing level l solve operator. Three of them represent the operations normally done with the solver operator, smoothing and residual formation. The other two represent restriction and interpolation. We have to approximate these with the solve operator, as at the stage in the setup where we have to make the decision, the restriction and interpolation operators have yet to be formed. To determine T_{switch}, we use the model to predict the time of five matrix-vector multiplications with a redistributed solve operator and two all-gather operations, one to gather the problem data and one to gather the right-hand side. For the redistributed operator, we assume for the purposes of the model that the number of floating point operations and data sent per message are divided equally among the chunks, and that each chunk sends messages to every other chunk. We assume the network parameters needed for the model have already been measured; we measure the computation rate from the time spent running 10 MatVecs with the locally stored portion of the solve operator.

We assume the all-gather operation gathers the problem data over a binary tree and then broadcasts that data along the same tree. Using the baseline α-β model from our past work, where α is the communication start-up time and β is the time required to send one double-precision floating point entry, which we modify as needed to suit the architecture, the expression for the all-gather time is

$$T_{\text{allgather}} = 2\left\lceil \log_2 \frac{P}{C} \right\rceil \alpha + 2\frac{N}{C}\left(1 + \left\lceil \log_2 \frac{P}{C} \right\rceil\right)\beta,$$

where P is the number of processes, C is the number of chunks, and N is the number of unknowns. If $T_{\text{switch}} < T_{\text{noswitch}}$, we make the switch to chunks, unless gathering the data would cause the matrix or the vector to spill out of cache if it did not before gathering, in which case we do not switch, as spilling out of cache can have a major negative impact on performance. We currently assume C is the number of MPI tasks per node, though we will vary this parameter in the future to better tune it to the architecture.

Problem	Setup Times			Cycle Times		
7pt Laplace	64 Cores	512 Cores	4096 Cores	64 Cores	512 Cores	4096 Cores
No Chunks	1.04 s	4.32 s	10.76 s	50.8 ms	138.4 ms	179.2 ms
Chunks	0.76 s	1.40 s	3.76 s	34.4 ms	67.9 ms	114.7 ms
Speedup	1.37	3.08	2.86	1.48	2.04	1.56
27pt Stencil	64 Cores	512 Cores	4096 Cores	64 Cores	512 Cores	4096 Cores
No Chunks	1.14 s	4.20 s	9.87 s	74.7 ms	149.4 ms	186.7 s
Chunks	0.68 s	1.53 s	3.58 s	48.9 ms	77.7 ms	130.3 ms
Speedup	1.68	2.74	2.76	1.53	1.92	1.43
Convection-Diffusion	64 Cores	512 Cores	4096 Cores	64 Cores	512 Cores	4096 Cores
No Chunks	0.95 s	3.94 s	10.62 s	46.5 ms	119.0 ms	167.9 ms
Chunks	0.79 s	1.69 s	3.58 s	35.8 ms	82.7 ms	96.1 ms
Speedup	1.20	2.33	2.97	1.30	1.44	1.75
9pt Laplace	64 Cores	256 Cores	1024 Cores	64 Cores	256 Cores	1024 Cores
No Chunks	0.47 s	0.98 s	1.35 s	25.9 ms	41.6 ms	46.7 ms
Chunks	0.28 s	0.65 s	0.94 s	22.7 ms	35.6 ms	35.3 ms
Speedup	1.68	1.51	1.44	1.14	1.17	1.32
Rotated Anisotropy	64 Cores	256 Cores	1024 Cores	64 Cores	256 Cores	1024 Cores
No Chunks	0.47 s	0.98 s	1.83 s	38.7 ms	58.0 ms	78.2 ms
Chunks	0.32 s	0.71 s	1.14 s	26.4 ms	40.6 ms	54.7 ms
Speedup	1.47	1.38	1.60	1.46	1.43	1.43

TABLE I
RESULTS OF CHUNKS VS. ORIGINAL ALGORITHM FOR EACH TEST PROBLEM.

IV. RESULTS

We tested our algorithm on Hera, a multicore Linux cluster at Lawrence Livermore National Laboratory. Hera consists of 800 compute nodes, with four quad-core 2.3 GHz AMD Opteron processors per node and a cache size of 512 KB per core. The nodes are connected by a DDR Infiniband interconnect organized as a two-level fat-tree. The network parameters needed for the performance model were measured by microbenchmarking and/or calculated from the network topology. More detail is available in [15].

For our test problems, we used a 3D Laplacian with a 7-point and a 27-point stencil, a non-symmetric convection-diffusion problem, a 2D Laplacian with a 9-point stencil and a 2D problem with an anisotropy of 0.01 rotated by 60 degrees. The 3D problems were run with $30 \times 30 \times 30$ points per core and run on 64, 512, and 4096 cores. The 2D problems were run with 150×150 points per core and run on 64, 256, and 1024 cores. The core counts were chosen so that the global problem size doubles in each dimension with each increase in core count. For each problem, we ran a combination of one AMG setup and 10 V-cycles ten times, averaging the reported times to avoid impact caused by noise. We used a cyclic mapping of MPI tasks per node, and the cyclic collective and point-to-point communicators for the chunks algorithm, keeping communication during the chunks stage entirely within nodes.

Table I presents the results for the original algorithm (No Chunks), the chunks algorithm with the performance model used to guide when to make the switch, and the speedup achieved. We achieved substantial speedups for all 3-dimensional cases, almost always over 40%. The setup phase in particular showed great improvement, with most speedups well over 2x, and much improved weak scalability. We also saw improvements for the 2-dimensional cases, which generally have much lower computational and communication complexities and therefore there is less potential for performance improvement.

V. Conclusions

We have successfully introduced a new algorithm that improves the performance of AMG through the reduction of data movement. The algorithm uses redundant data distribution to reduce and regularize the communication pattern of AMG, and enables communication to take place within well-defined and localized units of the machine. We have coupled our algorithm with a performance model that can decide dynamically at runtime when switching to the new algorithm is beneficial. In our experiments, we achieved speedups of up to 3x in the setup phase and 2x in the solve phase.

Current trends in computer architecture forecast an increasing role for our algorithm in preparing AMG for larger-scale machines. Per-chip and per-node core counts are expected to increase, with some researchers [17] envisioning a future with thousands of cores per chip. The chunks algorithm enables adaptation to this future. As the size of the problems being solved on larger machines increases, so can the number of chunks, to keep the problem size per core from being too large, while still keeping interprocess communication entirely within chips or nodes. In addition, the algorithm can be extended in a tree-like fashion by applying the simple chunks algorithm recursively to each chunk.

There are many avenues for future work. We are interested in the potential redundancy brings for resiliency and further algorithmic innovation. We expect to use the extra data to recover more quickly from faults. Furthermore, we will explore extensions to this approach that enable us to perform different computations on the duplicated pieces of data in parallel and then recombine them in a way that accelerates convergence. This idea was once popular [18], but was later found not to have enough benefit to overcome the cost of the extra work [19]. However, with the chunks algorithm able to place the duplicated pieces in places of the machine where work on each piece will not interfere with work on other pieces, we are hoping to see some form of performance gain from such an approach. Additionally, power is a growing concern and the reduction in data movement realized by this work is a significant first step to reducing power requirements. We will explore this issue further and investigate how to tune the redundancy to reduce power usage further without a loss in performance.

Another interesting topic is the interaction of the chunks algorithm with changes to the programming model. For simplicity, we considered an MPI-only programming model in this paper, but hybrid models that combine MPI with other programming models are becoming increasingly popular, most notably hybrid MPI/OpenMP for multicore clusters. The use of such a programming model has shown significant performance gains as well as increased scalability on several architectures [20], but will not entirely address the performance challenges it faces on coarse grids, particularly when dealing with systems at extreme scales. The combination of the chunks algorithm, which addresses the coarse grid difficulties, and a hybrid programming model, is therefore very promising. In summary, the chunks algorithm and the future research opportunities it opens have significant potential for ensuring the scalability of AMG to next generation parallel machines.

Acknowledgements

This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy award DE-SC0004131, and performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Neither Contractor, DOE, or the U.S. Government, nor any person acting on their behalf: (a) makes any warranty or representation, express or implied,

with respect to the information contained in this document; or (b) assumes any liabilities with respect to the use of, or damages resulting from the use of any information contained in this document.

REFERENCES

[1] R. D. Falgout, "An introduction to algebraic multigrid," *Computing in Science and Engineering*, vol. 8, pp. 24–33, 2006.

[2] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "Scaling hypre's Multigrid Solvers to 100,000 Cores," in *High-Performance Scientific Computing: Algorithms and Applications*, M. W. Berry, K. A. Gallivan, E. Gallopoulos, A. Grama, B. Philippe, Y. Saad, and F. Saied, Eds. Springer, 2012, pp. 261–279.

[3] V. E. Henson and U. M. Yang, "BoomerAMG: A parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, pp. 155–177, April 2002.

[4] "*hypre*: High performance preconditioners," http://www.llnl.gov/CASC/hypre/.

[5] H. De Sterck, U. M. Yang, and J. J. Heys, "Reducing complexity in parallel algebraic multigrid preconditioners," *SIAM Journal on Matrix Analysis and Applications*, vol. 27, pp. 1019–1039, 2006.

[6] H. De Sterck, R. D. Falgout, J. W. Nolting, and U. M. Yang, "Distance-two interpolation for parallel algebraic multigrid," *Numerical Linear Algebra With Applications*, vol. 15, pp. 115–139, April 2008.

[7] U. M. Yang, "On long-range interpolation operators for aggressive coarsening," *Numerical Linear Algebra With Applications*, vol. 17, pp. 453–472, April 2010.

[8] W. Gropp, "Parallel Computing and Domain Decomposition," in *Fifth Conference on Domain Decomposition Methods for Partial Differential Equations*, T. Chan, D. Keyes, G. Meurant, J. Scroggs, and R. Voigt, Eds. SIAM, 1992, pp. 349–361.

[9] A. H. Baker, R. D. Falgout, H. Gahvari, T. Gamblin, W. Gropp, K. E. Jordan, T. V. Kolev, M. Schulz, and U. M. Yang, "Preparing Algebraic Multigrid for Exascale," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-533076, March 2012.

[10] D. E. Womble and B. C. Young, "A model and implementation of multigrid for massively parallel computers," *International Journal of High Speed Computing*, vol. 2, pp. 239–255, 1990.

[11] K. Nakajima, "New Strategy for Coarse Grid Solvers in Parallel Multigrid Methods using OpenMP/MPI Hybrid Programming Models," in *2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, New Orleans, LA, February 2012, pp. 93–102.

[12] M. W. Gee, C. M. Siefert, J. J. Hu, R. S. Tuminaro, and M. G. Sala, "ML 5.0 Smoothed Aggregation User's Guide," Sandia National Laboratories, Tech. Rep. SAND2006-2649, February 2007.

[13] R. S. Sampath and G. Biros, "A parallel geometric multigrid method for finite elements on octree meshes," *SIAM Journal on Scientific Computing*, vol. 32, pp. 1361–1392, 2010.

[14] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp, "Modeling the Performance of an Algebraic Multigrid Cycle on HPC Platforms," in *25th ACM International Conference on Supercomputing*, Tucson, AZ, June 2011, pp. 172–181.

[15] H. Gahvari, W. Gropp, K. E. Jordan, M. Schulz, and U. M. Yang, "Modeling the Performance of an Algebraic Multigrid Cycle Using Hybrid MPI/OpenMP," in *41st International Conference on Parallel Processing*, Pittsburgh, PA, September 2012.

[16] ——, "Performance Modeling of Algebraic Multigrid on Blue Gene/Q: Lessons Learned," in *3rd International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, Salt Lake City, UT, November 2012.

[17] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, December 2006.

[18] P. O. Frederickson and O. A. McBryan, "Parallel Superconvergent Multigrid," in *Multigrid Methods: Theory, Applications, and Supercomputing*, S. F. McCormick, Ed. Marcel Dekker, 1988, pp. 195–210.

[19] L. R. Matheson and R. E. Tarjan, "Parallelism in Multigrid Methods: How Much Is Too Much?" *International Journal of Parallel Programming*, vol. 24, pp. 397–432, 1996.

[20] A. H. Baker, T. Gamblin, M. Schulz, and U. M. Yang, "Challenges of Scaling Algebraic Multigrid across Modern Multicore Architectures," in *25th IEEE Parallel and Distributed Processing Symposium*, Anchorage, AK, May 2011, pp. 275–286.